MODULE III

- **Assembler design options:**
  - **Machine Independent assembler features**
    - Literals
    - Symbol Defining Statements
    - Expressions
    - **Program blocks**
    - **Control sections**
  - **Assembler design options**
    - **Algorithm for Single Pass assembler**
    - **Multi pass assembler**
  - **Implementation example of MASM Assembler**

- **<u>Machine Independent assembler features</u>**
  - Following are the features which do not depend on the architecture of the machine.
    - Literals
    - Symbol Defining Statements
    - Expressions
    - Program blocks
    - Control sections

  - **<u>Literals</u>**
    - Programmers can be able to write the value of a constant operand as a part of the instruction. Such an operand is called **literals**.
    - A literal is defined with a prefix =
    - Eg: LDA =X'05'

    - **Literals vs Immediate Operand**
      - **Literals**
        - In case of literals the assembler generates the specified value as a constant at some other memory location
        - Target Address(TA) is the address of this generated constant.
        - The addressing mode of this instruction is either PC-relative or base-relative.
        - Eg:

```
45   001A ENDFIL LDA   =C'EOF'            032010
                                    nixbpe  disp
                             000000 110010  010
93                     LTORG
     002D *            =C'EOF'            454F46
215  1062 WLOOP   TD   =X'05'             E32011
230  106B         WD   =X'05'             DF2008
     1076 *            =X'05'             05
```

- In the above example EOF is stored in location(002D)
- Consider the following statement in the above program

  ENDFILL LDA =C'EOF'
- It has a 3-byte operand whose value is a character string EOF.
- This instruction follows Program Counter Relative addressing mode.
- TA= Address of the operand = (002D)
- After executing this instruction PC = (001D)
- Hence the displacement = TA - PC = (002D) - (001D)= (010)
- Therefore, the object code for this instruction is 032010

- Consider the following statement in the above program

  WLOOP TD =X'05'
- It has a 1-byte operand with hexadecimal value 05.
- This instruction follows Program Counter Relative addressing mode.
- TA= Address of the operand = (1076)
- After executing this instruction PC = (1065)
- Hence the displacement = TA - PC = (1076) - (1065)= (011)
- Therefore, the object code for this instruction is E32011

- **Immediate Operand**
  - In immediate mode the operand value is assembled as part of the instruction itself.
  - Eg:

    ```
    0020        LDA   #03      010003
    ```
  - We can have literals in SIC, but immediate operand is only valid in SIC\XE

- **Literal Pools**
  - All the literal operands used in a program are gathered together into one or more *literal pool*s.
  - There are two ways to place the literals in the program
    - Can place the literals at the end of the program (After END statement).
    - Can place the literals at some other location in the object program.
      - **Reason**: keep the literal operand close to the instruction
      - An assembler directive LTORG is used.
      - Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program or since the previous LTORG.
      - It is better to place the literals close to the instructions.
        - If the literal operand would be placed too far away from the instruction referencing, we cannot use PC-relative addressing or Base-relative addressing to generate Object Program. Here we are forced to choose extended instruction format. To avoid this we can use LTORG in different places in the program.

- **Literal Table (LITTAB)**
  - A literal table is a data structure created for the literals which are used in the program.
  - The literal table contains the ***literal name, operand value, length and address***.

- LITTAB is often organized as a hash table, using the literal name or value as the key

LITTAB

Literal	Hex Value	Length	Address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

- **Implementation of Literals**
  - **During Pass-1:**
    - The literal encountered is searched in the literal table.
    - If the literal already exists, no action is taken.
    - If it is not present, the literal name, operand value and length are added to the LITTAB.
    - When encounters a LTORG statement or the end of the program
      - The assembler makes a scan of the LITTAB and assigns an address for each literal not yet assigned an address.
      - Update the location counter value.
  - **During Pass-2:**
    - Search LITTAB for each literal operand encountered
    - Literal values placed at correct locations in the object program.
    - If the literal value represents an address in the program, the assembler must also generate the appropriate Modification Record.

- Allow literals that refer to the current value of the location counter.
  - '*' denotes a literal refer to the current value of program counter
  - Eg: **LDB =***
- **Duplicate literals**
  - The same literal used more than once in the program
  - e.g.

```
WLOOP   TD    =X'05'
----    ----- -----
----    ----- -----
        WD    =X'05'
```

  - The assemblers should recognize duplicate literals and store only one copy of the specified data value

Symbol Defining Statements

- **EQU Statement**
  - EQU is an assembler directive
  - It allows the programmer to define symbols and specify their values
  - Syntax: **Symbol EQU value**
  - The value can be a constant or an expression involving constants and any other symbol which is already defined.
    - Eg:

```
A   EQU  10
B   EQU  X-Y
```

- Usage:
  - To improve readability in place of numeric values
    - Eg: Replace **+LDT #4096**
      With

```
MAXLEN   EQU    4096
         +LDT   # MAXLEN
```

  - To define mnemonic names for registers
    - Eg: Replace **RMO 0,1**
      with

```
A    EQU  0
X    EQU  1
     RMO  A,X
```

- No forward reference
  - One restriction with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be predefined.
  - Eg:

✓

```
ALPHA    RESW    1
BETA     EQU     ALPHA
```

✗

```
BETA     EQU     ALPHA
ALPHA    RESW    1
```

- **ORG Statement**
  - ORG is an Assembler directive
  - Allow the assembler to reset the PC to values
  - Syntax: **ORG value**
  - When ORG is encountered, the assembler resets its LOCCTR to the specified value
  - ORG will affect the values of all labels defined until the next ORG
  - We can return to the normal use of LOCCTR by simply write ORG
  - ORG is used to control assignment storage in the object program.
  - No forward reference is allowed
    - All symbols used to specify the new LOCCTR value must have been previously defined.

✗

```
         ORG     ALPHA
BYTE1    RESB    1
BYTE2    RESB    1
BYTE3    RESB    1
         ORG
ALPHA    RESW    1
```

    - During pass1 assembler would not know what value to assign to the location counter in response to the first ORG statement. As a result, the symbols BYTE1, BYTE2 and BYTE3 could not be assigned during pass 1.

- <u>**Expressions**</u>
  - The assemblers allow the expressions as operand
  - The assembler evaluates the expressions and produces a single operand address or value
  - Expressions consist of

- Operator: +,-,*,/
- Constants
- User-defined symbols
- Special terms: *, the current value of LOCCTR
- Examples

```
MAXLEN    EQU   BUFEND-BUFFER
STAB      RESB  (6+3+2)*MAXENTRIES
BUFEND    EQU   *
```

The current value of location counter is assigned to BUFEND.

- Values of terms can be classified as absolute or relative.
  - Absolute terms
    - Independent of program location
    - Eg: Constants

      MAXLEN EQU 1000
  - Relative terms
    - Defined relative to the beginning of the program
    - Eg:
      - Labels on instructions
      - References to location counter: *
- Expressions can be either absolute or relative
  - Absolute Expression
    - Expression contains only absolute terms

      MAXLEN EQU 1000+5
    - Relative terms in pairs with opposite signs for each pair

      MAXLEN EQU BUFEND-BUFFER
      - BUFEND and BUFFER both are relative terms, representing addresses within the program. The expression BUFEND-BUFFER represents an absolute value.
      - When relative terms are paired with opposite signs, the dependency on the program starting address is canceled out. The result is an absolute value.
      - No relative term may enter into a multiplication or division operation.
  - Relative Expression
    - Contains an odd number of relative terms, with one more positive term than negative term.

      STAB EQU OPTAB + (BUFEND – BUFFER)
    - No relative term may enter into a multiplication or division operation.
      - Eg: 3*BUFFER is incorrect.
  - Expressions that are neither absolute nor relative will lead to assembler error.
    - Eg:
      - BUFEND+BUFFER
      - 100-BUFFER
      - 3*BUFFER
- Defining Symbol Types in the Symbol Table
  - To find the type of expression, we must keep track of the types of all symbols defined in this program.

- For this purpose we need a flag in the SYMTAB to indicate type of value (absolute or relative) in addition to the value itself.

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

- With this information the assembler can easily determine the type of each expression used as an operand and generate Modification Record in the object program for relative values.

- **<u>Program blocks</u>**
  - The source programs logically contained subroutines, data area etc.
  - Within the object program the generated machine instructions and data appeared in the same order as they were written in the source program.
  - Program blocks allow the generated machine instructions and data to appear in a different order while they are loading in memory.
    - Separating blocks for storing code, data, stack, and larger data block
  - Assembler directive: USE
    - Syntax: **USE [blockname]**
    - USE indicates which portion of the source program belongs to the various blocks.
    - At the beginning, statements are assumed to be part of the default block
    - If no USE statements are included, the entire program belongs to this single block
    - Each program block may actually contain several separate segments of the source program
  - Assembler rearrange these segments to gather together the pieces of each block and assign address
    - Separate the program into blocks in a particular order
    - Large buffer area is moved to the end of the object program
    - Program readability is better if data areas are placed in the source program close to the statements that reference them.
    - Consider the following program. Here 3 blocks are used.
      - Unnamed (default) block (block no: 0): contains the executable instructions of the program.
      - CDATA block (block no: 1): contains all data areas that are less in length.
      - CBLKS block (block no: 2): contains all data areas that consist of larger blocks of memory
    - At the beginning, statements are assumed to be part of the default block
    - The USE statement on line 92 signals the beginning of the block named CDATA.
    - The USE statement on line 103 signals the beginning of the block named CBLKS.

- The USE statements on line 123 and 208 resume the default block, and the statements on line 183 and 252 resume CDATA block.
- Line 107 is shown without a block number because the value of MAXLEN is an absolute symbol.

Line	Loc/Block		Source statement			Object code
5	0000	0	COPY	START	0	
10	0000	0	FIRST	STL	RETADR	172063
15	0003	0	CLOOP	JSUB	RDREC	4B2021
20	0006	0		LDA	LENGTH	032060
25	0009	0		COMP	#0	290000
30	000C	0		JEQ	ENDFIL	332006
35	000F	0		JSUB	WRREC	4B203B
40	0012	0		J	CLOOP	3F2FEE
45	0015	0	ENDFIL	LDA	=C'EOF'	032055
50	0018	0		STA	BUFFER	0F2056
55	001B	0		LDA	#3	010003
60	001E	0		STA	LENGTH	0F2048
65	0021	0		JSUB	WRREC	4B2029
70	0024	0		J	@RETADR	3E203F
92	0000	1		USE	CDATA	
95	0000	1	RETADR	RESW	1	
100	0003	1	LENGTH	RESW	1	
103	0000	2		USE	CBLKS	
105	0000	2	BUFFER	RESB	4096	
106	1000	2	BUFEND	EQU	*	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	

SUBROUTINE TO READ RECORD INTO BUFFER

Line	Loc/Block		Source statement			Object code
123	0027	0		USE		
125	0027	0	RDREC	CLEAR	X	B410
130	0029	0		CLEAR	A	B400
132	002B	0		CLEAR	S	B440
133	002D	0		+LDT	#MAXLEN	75101000
135	0031	0	RLOOP	TD	INPUT	E32038
140	0034	0		JEQ	RLOOP	332FFA
145	0037	0		RD	INPUT	DB2032
150	003A	0		COMPR	A,S	A004
155	003C	0		JEQ	EXIT	332008
160	003F	0		STCH	BUFFER,X	57A02F
165	0042	0		TIXR	T	B850
170	0044	0		JLT	RLOOP	3B2FEA
175	0047	0	EXIT	STX	LENGTH	13201F
180	004A	0		RSUB		4F0000
183	0006	1		USE	CDATA	
185	0006	1	INPUT	BYTE	X'F1'	F1

SUBROUTINE TO WRITE RECORD FROM BUFFER

Line	Loc/Block		Source statement			Object code
208	004D	0		USE		
210	004D	0	WRREC	CLEAR	X	B410
212	004F	0		LDT	LENGTH	772017
215	0052	0	WLOOP	TD	=X'05'	E3201B
220	0055	0		JEQ	WLOOP	332FFA
225	0058	0		LDCH	BUFFER,X	53A016
230	005B	0		WD	=X'05'	DF2012
235	005E	0		TIXR	T	B850
240	0060	0		JLT	WLOOP	3B2FEF
245	0063	0		RSUB		4F0000
252	0007	1		USE	CDATA	
253				LTORG		
	0007	1	*	=C'EOF'		454F46
	000A	1	*	=X'05'		05
255				END	FIRST	

- **Pass 1**
  - A separate location counter for each program block
    - At the beginning of a block, LOCCTR is set to 0.
    - Save and restore LOCCTR when switching between blocks
  - Assign each label an address relative to the start of the block that contains it.
  - Store the block name (or number) in the SYMTAB along with the assigned relative address of the label
  - At the end of Pass1 the latest value of LOCCTR for each block indicates the length of that block.
  - At the end of Pass1 the assembler constructs a block table that contains the block name, block number, starting addresses and length of all blocks.

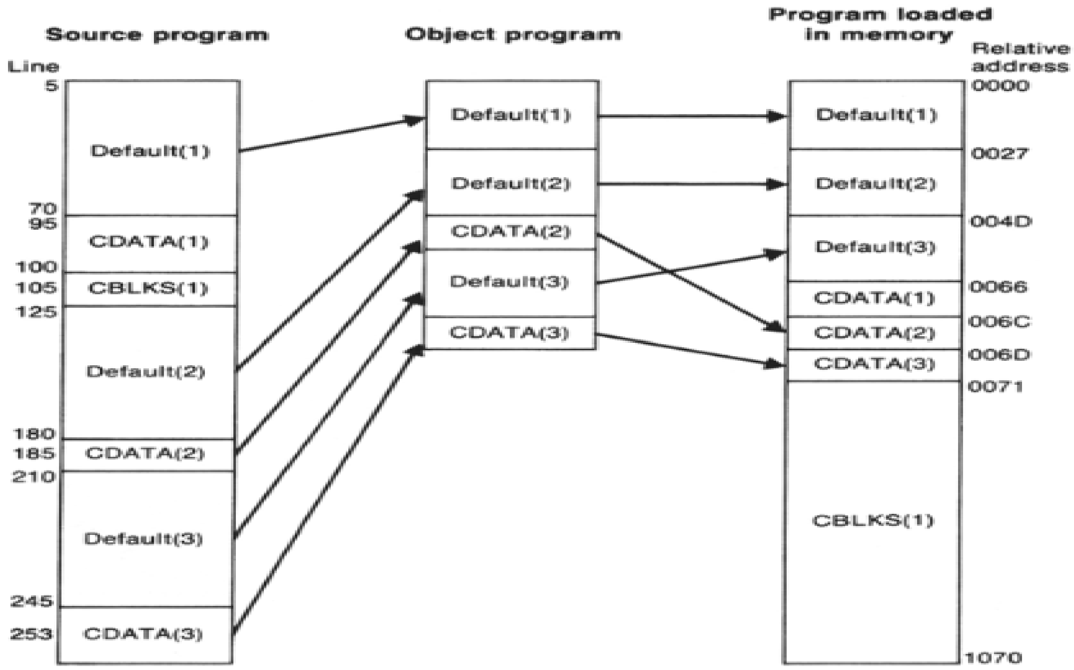
Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

- **Pass 2**
  - Calculate the address for each symbol relative to the start of the object program by adding the location of the symbol relative to the start of its block, to the assigned block starting address.

  - Eg:
    - Consider the instruction **LDA LENGTH**
    - The relative location of LENGTH in CDATA block = 0003
    - Starting address for CDATA = 0066
    - Therefore, TA = 0003 + 0066 = 0069
    - This instruction is to be assembled using PC-relative addressing mode.
    - After fetching this instruction, PC = 0009. Since the default block starts at location 0000, this address = 0000 + 0009 = 0009
    - Displacement = TA-PC = 0069 – 0009 = 0060
    - Therefore, the object code is 032060

```
H^COPY  ^000000^001071
T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEE^032055^0F2056^010003
T^00001E^09^0F2048^4B2029^3E203F                                       Default(1)
T^000027^1D^B410^B400^B440^75101000^E32038^332FFA^DB2032^A004^332008^57A02F^B850
T^000044^09^3B2FEA^13201F^4F0000                                       Default(2)
T^00006C^01^F1                                                         CDATA(2)
T^00004D^19^B410^772017^E3201B^332FFA^53A016^DF2012^B850^3B2FEF^4F0000 Default(3)
T^00006D^04^454F46^05                                                  CDATA(3)
E^000000
```

  - First 2 text records are generated from lines 5 through 70(default block 1).
  - No new text record is created for lines 95 through 105, because it is not generated any code. Next 2 text records come from lines 125 through 180(default block 2). Fifth text record is for CDATA 2 block and so on.



- The loader loads the default block in the memory from location 0000 to 0065. CDATA will occupy locations from 0066 through 0070. CBLKS will occupy locations 0071 through 1070.
- CDATA(1) and CBLKS(1) are not present in object program. Storage will automatically be reserved for these areas when the program is loaded.

▪ **Benefits of Program Blocks**
- Here the larger buffer area is moved to the end of the object program. So we can avoid the use of extended instruction format.
- Program readability is improves if the definition of data areas are placed in the source program close to the statements that reference them.

Pass 1 Algorithm

Begin
    block number = 0
    LOCCTR[i] = 0 for all i
    Read the first input line
    If *OPCODE = 'START'* then
    {
        Write line into intermediate file
        Read next input line
    }
    While *OPCODE != 'END'* do
    {   If *OPCODE = 'USE'* then
        {   If *there is no operand name* then   block name = Default
            Else   block name = OPERAND name
            If *there is no entry for block name in block table* then
                Insert (block name, block no++) in to block table
            i = bock number for block name
            if *there is not a comment line* then
            {   If *there is a symbol in the LABEL field* then

```
                {
                    Search SYMTAB for LABEL
                    If found then        Set error flag
                    Else        Insert (LABEL, LOCCTR[i], block number) into SYMTAB
                }
                Search OPTAB for OPCODE
                If found then                        LOCCTR[i] = LOCCTR[i] +3
                Else if OPCODE = 'WORD' then         LOCCTR[i] = LOCCTR[i] +3
                Else if OPCODE = 'RESW' then         LOCCTR[i] = +3 * #OPERAND
                Else if OPCODE = 'RESB' then         LOCCTR[i] = + #OPERAND
                Else if OPCODE = 'BYTES' then        LOCCTR[i]= +length of the constant
                Else                                 Set error flag
            }
            Write line into intermediate file
            Read next input line
        }
    }
    LENGTH[i] = LOCCTR[i] for all i
    Address[0] = starting address
    Address[i] = Address[i-1] + Length[i-1]     for all i=1 to max(block number)
    Insert (Address[i], Length[i]) in block table for all i
End
```

Pass 2 Algorithm

```
If OPCODE = 'USE' then
  set block number for block name with OPERAND field
  search SYMTAB for OPERAND
  store symbol value + address [block number] as operand address
end {Pass 2}
```

- <u>**Control sections**</u>
  - **Program blocks v.s. Control sections**
    - Program blocks: Segments of code that are rearranged within a single object program unit
    - Control sections: Segments of code that are translated into independent object program units
  - These are most often used for subroutines or other logical subdivisions of a program
  - The programmer can assemble, load, and manipulate each of these control sections separately
  - Assembler directive: **CSECT**
    - Syntax: **secname CSECT**
  - Separate location counter for each control section. Initial value of the location counter is 0.
  - Instructions in one control section may need to refer to instructions or data located in another section. Assembler has no idea where any other control sections will be located at execution time.

- It is necessary to provide some means of linking them together. For this purpose we can use the following 2 assembler directives
  - External definition **EXTDEF symbol1,symbol2, ... ,symboln**
    - Define symbols that are defined in this control section and may be used by other sections
    - Ex: EXTDEF BUFFER, BUFEND, LENGTH

  - External reference **EXTREF symbol1,symbol2, ... ,symboln**
    - Define symbols that are used in this control section and are defined elsewhere
    - Ex: EXTREF RDREC, WRREC
    - To reference an external symbol, extended format instruction is needed.

- The following program consist of 3 control sections
  - COPY: Main program. This section continues until the CSECT statement on line 109.
  - RDREC: Subroutine. This control section is from line no 109 to 190.
  - WRREC: Subroutine. This control section is from line no 193 to 255.
- Ex: Consider the instruction

  15 0003 CLOOP +JSUB RDREC
  - RDREC is an external reference.
  - The assembler has no idea where RDREC is
  - The assembler inserts an address of zero.
  - The proper address to be inserted at load time
  - Can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)
  - The object code is: 4B100000
  - The assembler generates information for each external reference that will allow the loader to perform the required linking.
- Ex: Consider the instruction

  160 0017 +STCH BUFFER,X
  - BUFFER is an external reference. The assembler has no idea where BUFFER is
  - The assembler inserts an address of zero
  - The object code is: 57900000
- Ex: Consider the instruction

  190 0028 MAXLEN WORD BUFEND-BUFFER
  - BUFEND and BUFFER are two eternal reference symbols.
  - Assembler inserts a value of 0
  - The object code is: 000000
  - When the program is loaded, the loader will add to this data area the address of BUFEND and subtract from it the address of BUFFER.
- Ex: Consider the instruction

  107 1000 MAXLEN EQU BUFEND-BUFFER
  - BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately
- Restriction

- Both terms in each pair of an expression must be within the same control section
    - Legal: BUFEND-BUFFER
    - Illegal: RDREC-COPY

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
6			EXTDEF	BUFFER,BUFEND,LENGTH	
7			EXTREF	RDREC,WRREC	
10	0000	FIRST	STL	RETADR	172027
15	0003	CLOOP	+JSUB	RDREC	4B100000
20	0007		LDA	LENGTH	032023
25	000A		COMP	#0	290000
30	000D		JEQ	ENDFIL	332007
35	0010		+JSUB	WRREC	4B100000
40	0014		J	CLOOP	3F2FEC
45	0017	ENDFIL	LDA	=C'EOF'	032016
50	001A		STA	BUFFER	0F2016
55	001D		LDA	#3	010003
60	0020		STA	LENGTH	0F200A
65	0023		+JSUB	WRREC	4B100000
70	0027		J	@RETADR	3E2000
95	002A	RETADR	RESW	1	
100	002D	LENGTH	RESW	1	
103			LTORG		
	0030	*	=C'EOF'		454F46
105	0033	BUFFER	RESB	4096	
106	1033	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	
109	0000	RDREC	CSECT		
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
122			EXTREF	BUFFER,LENGTH,BUFEND	
125	0000		CLEAR	X	B410
130	0002		CLEAR	A	B400
132	0004		CLEAR	S	B440
133	0006		LDT	MAXLEN	77201F
135	0009	RLOOP	TD	INPUT	E3201B
140	000C		JEQ	RLOOP	332FFA
145	000F		RD	INPUT	DB2015
150	0012		COMPR	A,S	A004
155	0014		JEQ	EXIT	332009
160	0017		+STCH	BUFFER,X	57900000
165	001B		TIXR	T	B850
170	001D		JLT	RLOOP	3B2FE9
175	0020	EXIT	+STX	LENGTH	13100000
180	0024		RSUB		4F0000
185	0027	INPUT	BYTE	X'F1'	F1
190	0028	MAXLEN	WORD	BUFEND-BUFFER	000000
193	0000	WRREC	CSECT		
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
207			EXTREF	LENGTH,BUFFER	
210	0000		CLEAR	X	B410
212	0002		+LDT	LENGTH	77100000
215	0006	WLOOP	TD	=X'05'	E32012
220	0009		JEQ	WLOOP	332FFA
225	000C		+LDCH	BUFFER,X	53900000
230	0010		WD	=X'05'	DF2008
235	0013		TIXR	T	B850
240	0015		JLT	WLOOP	3B2FEE
245	0018		RSUB		4F0000
255			END	FIRST	
	001B	*	=X'05'		05

- The assembler must include information in the object program that will cause the loader to insert proper values where they are required. Define Record; Refer Record and Modification Record are used for this purpose.

	Column	Contents
Define Record	1	D
	2-7	Name of external symbol defined in this control section
	8-13	Relative address of symbol within this control section (HEX)
	14-73	Repeat information in Col. 2-13 for other external symbols
Refer Record	1	R
	2-7	Name of external symbol referred to in this control section
	8-73	Names of other external reference symbols
Mod. Record	1	M
	2-7	Starting address of the field to be modified, relative to the beginning of the program (HEX)
	8-9	Length of the field to be modified, in half-bytes (HEX)
	10	Modification flag (+ or -)
	11-16	External symbol whose value is to be added to or subtracted from the indicated field

- The object program corresponding to the above is

```
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000
```

```
H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201F^E3201B^332FFA^DB2015^A004^332009^57900000^B850
T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND
M^000028^06^-BUFFER
E
```

```
H^WRREC ^000000^00001C
R^LENGTH^BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^3B2FEE^4F000005
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

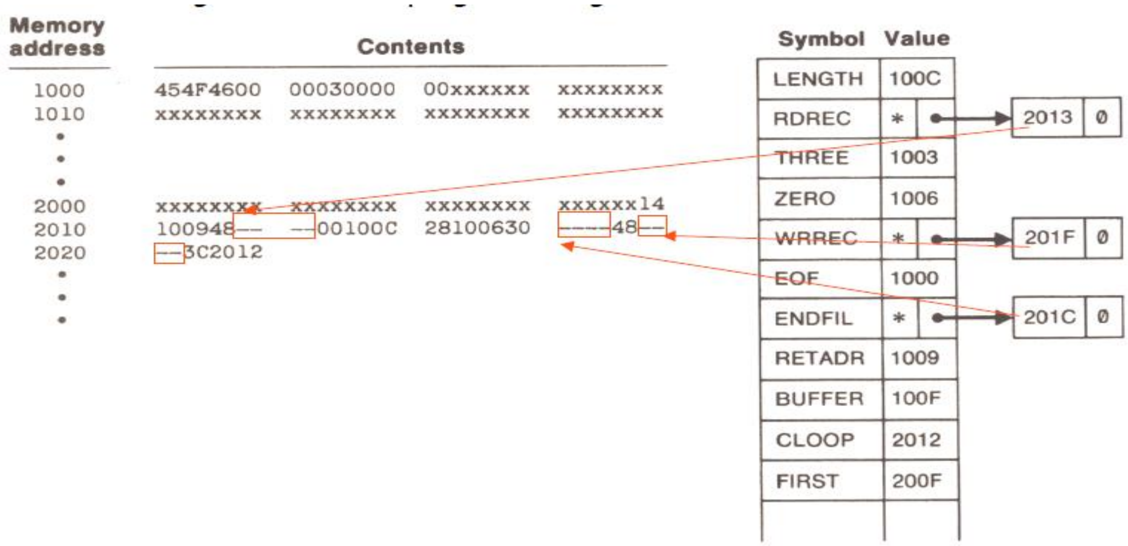
- **<u>Assembler Design Options</u>**
  - **Single Pass Assembler**
  - **Multipass Assembler**

  - **<u>Single Pass Assembler</u>**
    - The main problem in designing the assembler using single pass was to resolve forward references. There are two types of forward references.
      - Forward reference to data items
        - Solution
          - Define all the storage reservation statements at the beginning of the program rather at the end.
      - Forward jumping: Forward reference to labels on the instructions
        - Solution
          - Insert (label, *address_to_be_modified*) to SYMTAB
          - Usually, *address_to_be_modified* is stored in a linked-list
    - There are two types of one-pass assemblers:
      - **Load-and-go assemblers:**
        - Generates object code directly in memory for immediate execution.
        - No object program is written out, no loader is needed.
        - The actual address must be known at assembly time.
        - It is useful in a system with frequent program development and testing
        - Programs are re-assembled nearly every time they are run.
      - **Object Program Output Assembler:**
        - This assembler produces the usual kind of object code for later execution.
        - This assembler is used on systems where external working storage devices are not available.
    - **Load-and-go assemblers Algorithm**
      - When a forward reference is encountered
        - Omits the operand address if the symbol has not yet been defined
        - Enters this undefined symbol into SYMTAB and indicates that it is undefined
        - Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
      - When the definition for the symbol is encountered, scans the reference list and inserts the address.
      - At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols. Otherwise jump to the location specified in END statement.

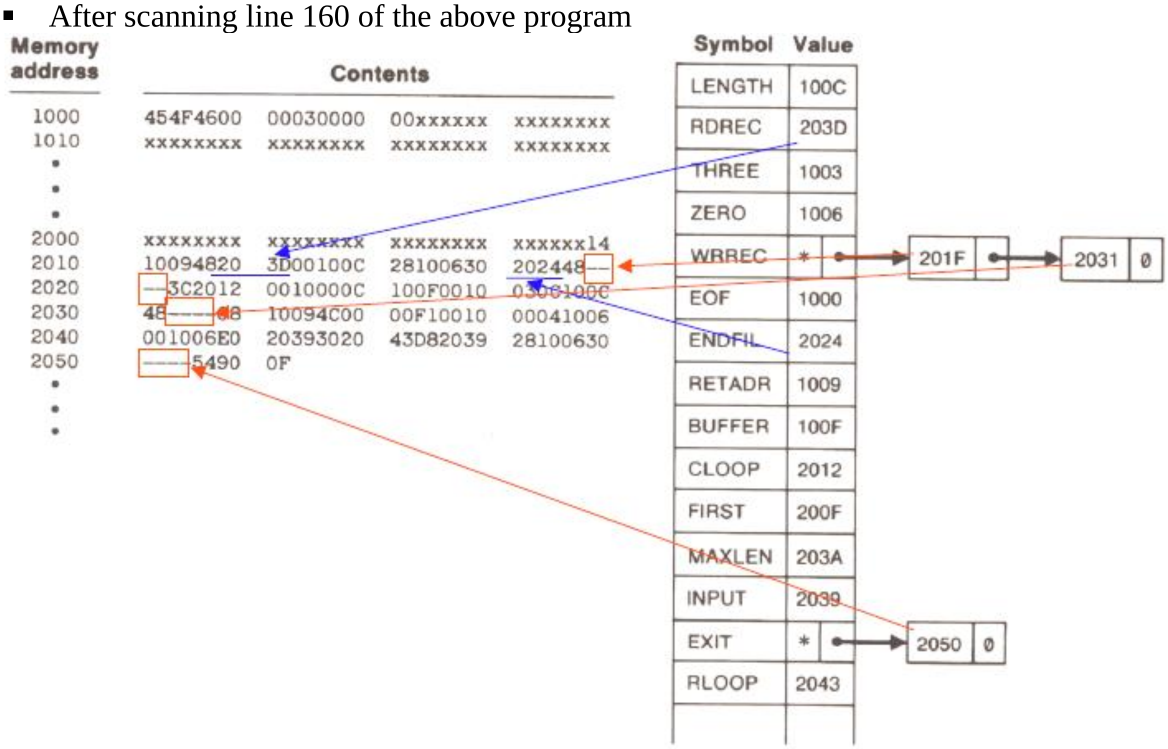
- The following program avoids forward data reference problem

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
SUBROUTINE TO READ RECORD INTO BUFFER					
121	2039	INPUT	BYTE	X'F1'	F1
122	203A	MAXLEN	WORD	4096	001000
124		.			
125	203D	RDREC	LDX	ZERO	041006
130	2040		LDA	ZERO	001006
135	2043	RLOOP	TD	INPUT	E02039
140	2046		JEQ	RLOOP	302043
145	2049		RD	INPUT	D82039
150	204C		COMP	ZERO	281006
155	204F		JEQ	EXIT	30205B
160	2052		STCH	BUFFER,X	54900F
165	2055		TIX	MAXLEN	2C203A
170	2058		JLT	RLOOP	382043
175	205B	EXIT	STX	LENGTH	10100C
180	205E		RSUB		4C0000
SUBROUTINE TO WRITE RECORD FROM BUFFER					
206	2061	OUTPUT	BYTE	X'05'	05
207		.			
210	2062	WRREC	LDX	ZERO	041006
215	2065	WLOOP	TD	OUTPUT	E02061
220	2068		JEQ	WLOOP	302065
225	206B		LDCH	BUFFER,X	50900F
230	206E		WD	OUTPUT	DC2061
235	2071		TIX	LENGTH	2C100C
240	2074		JLT	WLOOP	382065
245	2077		RSUB		4C0000
255			END	FIRST	

- In the main subroutine RDREC(line 15), WRREC(line 35, line 65) and ENDFILL(line 30) are forward references.
- After scanning line 40 of the above program



- The following symbols are not yet defined.
  - RREC is referred to the location 2013
  - ENDFIL is referred to the location 201F
  - WRREC is referred to the location 201C
- After scanning line 160 of the above program



- Some of the forward references have been resolved by this time, while others have been added.
- When the symbol ENDFILL was defined (line 45), the assembler places 2024 in the SYMTAB entry.
- Insert 2024 in the location 201C. Then delete the linked list.
- Similar operations are happened for all forward references.

- **Load-and-go Single Pass Assembler Algorithm**

```
Begin
    Read 1st input line
    If OPCODE = 'START' then
    {
          Starting address = #OPERAND
          LOCCTR = Starting address
          Read the next input line
     }
     Else
          LOCCTR = 0
     While OPCODE != 'END' do
     {
          If there is not a comment line then
          {
               If there is a symbol in the LABEL field then
               {
                    Search SYMTAB for LABEL
                    If found then
                    {
                         If symbol value as null then
                         {
                              Symbol value = LOCCTR
                              Search the linked list with corresponding operand
                              Generate operand addresses as corresponding to symbol value
                              Delete the linked list
                          }
                    }
                    Else
                           Insert (LABEL, LOCCTR) into SYMTAB
               }
               Search OPTAB for OPCODE
               If found then
               {
                    Search SYMTAB for OPERAND address
                    If found then
                    {
                         If symbol value != null then
                              OPERAND address = symbol value
                          Else
                    Insert a node at the end of the linked list with address as LOCCTR+1
                    }
                    Else
                    {     Insert (symbol name, null) into SYMTAB
                          Create a linked list with address as LOCCTR+1
                    }
                    Generate object code and load it in memory location LOCCTR
```

```
            LOCCTR = LOCCTR +3
        }
        Else if OPCODE = 'WORD' then
        {
            Object code = #OPERAND
            load this object code in memory location LOCCTR
            LOCCTR = LOCCTR +3
        }
        Else if OPCODE = 'RESW' then
            LOCCTR = LOCCTR +3x#OPERAND
        Else if OPCODE = 'RESB' then
            LOCCTR = LOCCTR + #OPERAND
        Else if OPCODE = 'BYTE' then
        {
            Convert constant to object code and load it in memory location LOCCTR
            LOCCTR = LOCCTR +length of the constant
        }
        Else
            Set error flag
    }
    Read the next input line
}
If there are still SYMTAB entries indicated undefined symbols
    Reports the error
Else
    Jump to the location specified in END statement.
End
```

- **Object Program Output Assembler**
  - Forward references are entered into SYMTAB as before.
  - When the definition of the symbol is encountered, the assembler generates another Text Record with the correct operand address of each entry in the linked list.
  - When the program is loaded, the incorrect address 0 will be updated by the Text Record containing the symbol definition.
  - The object program records must be kept in their original order when they are presented to the loader.
  - The object code for the above program is

```
H^COPY  ^001000^00107A
T^001000^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

- When the definition of ENDFIL on line 45 is encountered, the assembler generates the 3rd Text Record. This record specifies that the value 2024 is to be loaded at location 201C. When the program is loaded the value 2024 will replace the 0000 previously loaded.

- **Object Program Output Single Pass Assembler Algorithm**

```
Begin
    Read 1st input line
    If OPCODE = 'START' then
    {
        Starting address = #OPERAND
        LOCCTR = Starting address
        Read the next input line
    }
    Else
        LOCCTR = 0
    Create Header Record and write it to object program
    Initialize 1st Text Record
    While OPCODE != 'END' do
    {
        If there is not a comment line then
        {
            If there is a symbol in the LABEL field then
            {
                Search SYMTAB for LABEL
                If found then
                {
                    If symbol value as null then
                    {
                        Symbol value = LOCCTR
                        Generate separate Text record with corresponding operand address
                        of each entry in the linked list
                        Delete the linked list
                    }
                }
                Else
                    Insert (LABEL, LOCCTR) into SYMTAB
            }
            Search OPTAB for OPCODE
            If found then
            {
                Search SYMTAB for OPERAND address
                If found then
                {
                    If symbol value != null then
                        OPERAND address = symbol value
                    Else
                        Insert a node at the end of the linked list with address as LOCCTR+1
```

```
                }
                Else
                {   Insert (symbol name, null) into SYMTAB
                     Create a linked list with address as LOCCTR+1
                }
                Generate object code
                LOCCTR = LOCCTR +3
            }
            Else if OPCODE = 'WORD' then
            {
                LOCCTR = LOCCTR +3
                Object code = #OPERAND
            }
            Else if OPCODE = 'RESW' then
                LOCCTR = LOCCTR +3x#OPERAND
            Else if OPCODE = 'RESB' then
                LOCCTR = LOCCTR + #OPERAND
            Else if OPCODE = 'BYTE' then
            {
                LOCCTR = LOCCTR +length of the constant
                Convert constant to object code
            }
            Else
                Set error flag
            If object code will not fit into the current text record then
            {
                Write Text Record into object program
                Initialize new Text Record
            }
            Add object code to Text Record
        }
        Read the next input line
    }
    Write last Text Record to object program
    Write End Record to object program
End
```

- **Multipass Assembler**
  - The symbols used on the RHS of EQU should be defined previously in the program.
  - Eg:

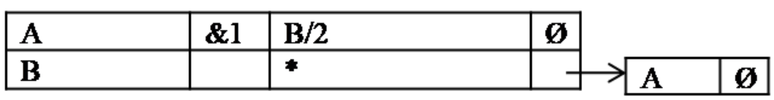
ALPHA	EQU	BETA
BETA	EQU	DELTA
DELTA	RESW	1

    - The symbol BETA cannot be assigned a value when it is encountered during Pass1 because DELTA has not yet been defined.
    - Hence ALPHA cannot be evaluated during Pass 2.
    - Symbol definition must be completed in pass 1.

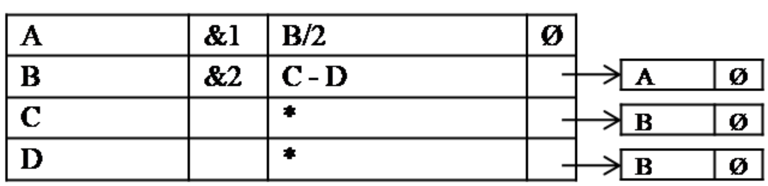
- Forward references tend to create difficulty for a person reading the program.
- The general solution for forward references is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.
- It is not necessary for such an assembler to make more than 2 passes over the entire program.
- The portions of the program that involve forward references in symbol definition are saved during Pass 1.
- Additional passes through these stored definitions are made as the assembly progresses.
- This process is followed by a normal Pass 2.
- **Implementation**
  - For a forward reference in symbol definition, we store in the SYMTAB:
    - The symbol name
    - The defining expression
    - The number of undefined symbols in the defining expression
  - The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
  - When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.
- Example

```
1    A    EQU   B/2
2    B    EQU   C-D
3    E    EQU   D-1
4    D    RESB  4096
5    C    EQU   *
```

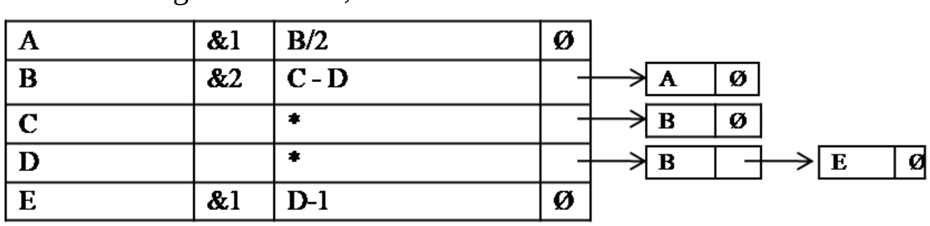
  - After executing statement 1, the SYMTAB will become



    - &1 represent the number of undefined symbols in the defining expression
    - B/2 is the defining expression
    - * indicate the undefined symbol
    - The node A represents depending list.
  - After executing statement 2, the SYMTAB will become



  - After executing statement 3, the SYMTAB will become



- Suppose the address of D is 1034. After executing statement 4, the SYMTAB will become

A	&1	B/2	Ø	
B	&1	C - 1034		→ A | Ø
C		*		→ B | Ø
D		1034	Ø	
E		1033	Ø	

- After executing statement 5, C will be LOCCTR. The SYMTAB will become

A		800	Ø
B		1000	Ø
C		2034	Ø
D		1034	Ø
E		1033	Ø

- **<u>Implementation example of MASM Assembler</u>**
  - An MASM assembler program is written as a collection of segments.
  - Commonly used segments are CODE, DATA, CONST and STACK.
  - Segments are addressed via segment registers
  - These registers are automatically set by the system loader when a program is loaded for execution.
    - CODE segment → CS register
      - If CS is set, then the current segment contains the label specified in the END statement.
    - STACK → SS register
      - SS is set indicate the last stack segment is processed by the loader.
    - DATA and CONST → DS, ES, FS or GS registers
      - If the programmer does not specify a segment register, one is selected by the assembler.
      - Default register is DS.
      - This can be changed by using ASSUME assembly directive

        ASSUME ES:DATASEG2

        ES indicates the segment DATASEG2. Any references to labels that are defined in DATASEG2 will be assembled using register ES
  - Jump instructions are assembled in 2 different ways
    - Near Jump
      - The target will be in the same code segment
      - It is assembled using the current code segment register CS
      - Instruction size may be 2 or 3 bytes
    - Far Jump
      - The target will be in a different code segment
      - It is assembled using a different segment register, which is specified in an instruction prefix.
      - Instruction size is 5 bytes
  - Forward references to a label in a source program can cause problems:

- Eg: JMP TARGET
  - If the definition of TARGET occurs in the program before JMP instruction, the assembler can tell whether this is a near jump or far jump. It is not possible in the case of forward jump.
  - By default, MASM assumes that a forward jump is a near jump.
  - If the target of the jump is in another code segment, the programmer must warn the assembler by writing JMP FAR PTR TARGET
  - If the jump address is within 128 bytes, the programmer can specify a shorter(2 bytes) near bytes by writing JMP SHORT TARGET
- Length of the assembled instruction is depends on its operand
  - Eg: operands of ADD instruction can be
    - Registers
    - Memory locations: May take varying amount of space, depending upon the location of the operand.
    - Immediate operands: May occupy from 1 to 4 bytes in the instruction
- Pass 1 of an x86 assembler is more complex than Pass 1 of SIC assembler
  - During Pass 1 of x86
    - Analyze the operands of each instruction
    - Looking at the operation code table
      - It contains information on which addressing modes are valid for each operand.
- Segments in a MASM source program can be written in more than one part.
  - All the parts are gathered together by the assembly process.
- References between segments are handled by the assembler.
  - Use the directive PUBLIC. It has the same function as EXTDEF in SIC/XE.
- External references between separately assembled modules must be handled by the linker.
  - Use the directive EXTRN. It has the same function as EXTREF in SIC/XE.
- The object program from MASM may be in several different formats
  - Allow easy and efficient execution of the program in a variety of operating environments.
- MASM produces an instruction timing that shows the number of clock cycles required to execute each instruction.

Previous Year University Questions

1. What is a Literal? How is a literal handled by an assembler?
2. With example, write notes on Program Blocks.
3. How the assembler handles multiple Program blocks?
4. What are control sections? What is the advantage of using them?
5. What are control sections? Illustrate with an example, how control sections are used and linked in an assembly language program.
6. Explain the format and purpose of Define and Refer records in the object program.
7. What are the uses of assembler directives EXTDEF and EXTREF?
8. How are control sections different from program blocks? Explain, with proper examples, the purpose of EXTREF and EXTDEF assembler directives.
9. Give the format and purpose of the different record types present in an object program that uses multiple control sections

10. Develop the records (excluding header, text and end records) for the following control section named COPY

Loc	Source Statement		
0000	COPY	START	0
		EXTDEF	BUFFER, BUFFEND, LENGTH
		EXTREF	RDREC,WRREC
0000	FIRST	STL	RETADR
0003	CLOOP	+JSUB	RDREC
0007		LDA	LENGTH
000A		COMP	#0
000D		JEQ	ENDFIL
0010		+JSUB	WRREC
0014		J	CLOOP
0017	ENDFIL	LDA	=C 'EOF'
001A		STA	BUFFER
001D		LDA	#3
0020		STA	LENGTH
0023		+JSUB	WRREC
0027		J	@RETADR
002A	RETADR	RESW	1
002D	LENGTH	RESW	1
		LTORG	
0030	*	=C 'EOF'	
0033	BUFFER	RESB	4096
1033	BUFEND	EQU	*
1000	MAXLEN	EQU	BUFEND-BUFFER

11. Explain how external references are handled by an assembler.
12. Distinguish between Program Blocks and Control Section
13. Differentiate between control sections and program blocks with the help of an example.
14. Differentiate Program Blocks and Control Sections. Explain how address calculation is performed in the case of Program Blocks
15. What is a load and go assembler?
16. Explain the concept of single pass assembler with a suitable example.
17. Explain the working of any one type of One pass Assembler
18. What is a forward reference? How are forward references handled by a single pass assembler?
19. Explain the working of Multi pass assemblers with an example.
20. Employ multipass assembler to evaluate the following expressions

Expression No.	Loc	Source Statement		
1		HALFSZ	EQU	MAXLEN/2
2		MAXLEN	EQU	BUFEND-BUFFER
3		PREVBT	EQU	BUFFER-1
4	4034	BUFFER	RESB	4096
5	5034	BUFEND	EQU	*

21. Write short notes on MASM assembler.